САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
Математико-механический факультет

Кафедра Информатики

Шмагринский Игорь Олегович

Анализ возможности и эффективности параллельной реализации алгоритма packJPG

Бакалаврская работа

Научный руководитель:
к. ф.-м. н., доцент Н.Ю. Ловягин

подпись

Рецензент:
д. ф.-м. н., доцент Т.О. Евдокимова

подпись

Санкт-Петербург
2017

SAINT-PETERSBURG STATE UNIVERSITY
Mathematics & Mechanics Faculty

Department of Computer Science

Igor Shmagrinsky

Analysis of the possibility and efficiency of the parallel implementation of the packJPG algorithm

Bachelor's Thesis

Scientific supervisor:
docent N.Y. Lovyagin

signature

Reviewer:
docent N.N. Neizvestny

signature

Saint-Petersburg
2017

Оглавление

Введение

Быстрое развитие IT технологий позволило в настоящее время даже в мобильных устройствах иметь более чем один процессор, что способствует более быстрому выполнению трудоемких операций при условии использования алгоритмов параллельных вычислений. Однако, еще далеко не все программы и алгоритмы адаптированы для многопроцессорных (многоядерных) архитектур, такие программы работают неэффективно, так как не используют всю мощь современных устройств. Поэтому задача адаптации алгоритмов для параллельных вычислений является важной и актуальной.

В данной работе расмотрен один из самых эффективных алгоритмов сжатия изображений без потерь PackJPG, который уступает другим алгоритмам этой области лишь по времени работы. Было проведено исследование возможности адаптации данного алгоритма и его реализация для многопроцессорной архитектуры с помощью технологии OpenMP. Выполнена сравнительная оценка времени работы данного алгоритма и его параллельной модификации.

Актуальность работы обусловлена еще и тем, что данный алгоритм дает существенную экономию объема JPEG-изображения (до 26%) без потери качества, что недоступно при использовании архиваторов, но требует значительных затрат времени на сжатие и распаковку.Скорость кодирования составляет 600 килобайт в секунду. Параллельные вычисления позволили бы сократить это время в несколько раз, доведя до приемлемых величин для небольших изображений. Сложность исследования возможности параллельной реализации алгоритма связана с тем, что спецификация программы и математическое описание недоступно в открытой печати , фактически данная работа выполнялась путем анализа алгоритма по исходному коду.

1. Предварительные сведения

Сжатие — это техника направленная на понижение объемаП

Существует множество разнообразных техник и стандартов для сжатия мультемедийных данных с потерями. Одним из таких стандартов на протяжении многих лет является формат изображений JPEG (ISO/IEC 10918).

1.1. Стандарт сжатия изображений JPEG

JPEG—это стандарт сжатия изображений разработаный Joint Photogrphic Experts Group. Он был официально одобрен мировым сообществом в 1992 году.Процесс сжатия, осуществляемый алгоритмом, стандарта JPEG состоит из приведенных ниже основных процедур.

1.1.1. Кодирование

Хотя *JPEG* файл может быть закодирован разными путями, но самым известным и популярным является *JFIF* кодировние. Процесс кодирования состоит из нескольких шагов.

Преобразование цветового пространства

Основны цвета изображения можно представить с помощью цветового пространства *RGB*. Такое представление, однако, сильно коррелирует, что подразумевает, что это цветовое пространство не очень подходит для независимого кодирования. Так как человеческая зрительная система менее чувствительна к позиции и движению ярких цветов. Следовательно, целесообразней изпользовать цветовое пространство *YCrBr*.

Разбиение исходного изображения

Изображение разбивается на блоки размера 8x8 пикселей, с каждым из которых ведется дальнейшая работа.

Дискретное косинусное преобразование

Дальше каждая компонента *(Y,Cr,Br)* каждого блока преобразуется в частотную форма. Для этого используется двумерное дискретное косинусное преобразование второго типа. Перед вычислением этого преобразования все значения сдвигаются из положительного интервала $[0, 255]$в интервал $[- - -128, 127]$ вычитанием из каждого значения компоненты блока 128. Это действие является обязательным, так как та-

кой интервал значений является одним из требований для дискретного косинусного преобразования. В результате будет получен блок $g_{x,y}$, с которым мы будем работать дальше.

Блок g преообразуется по следующему принципу:

$$G_{u,v} = \frac{1}{4}\alpha(u)\alpha(v)\sum_{x=0}^{7}\sum_{y=0}^{7} g_{x,y}\cos\left[\frac{(2x+1)u\pi}{16}\right]\cos\left[\frac{(2y+1)v\pi}{16}\right],$$

где:

- u — вертикальная пространственная частота для целых чисел $0 \leq u < 8$
- v — горизонатальная пространственная частота для целых чисел $0 \leq v < 8$
- $\alpha(u) = \begin{cases} \frac{1}{\sqrt{2}}, & \text{if } u = 0 \\ 1, & \text{иначе} \end{cases}$ — норма, необходимая для того чтобы преобразование было ортомнормированным.
- $g_{x,y}$ — это значение которое содержит в себе пиксель с координатами (x, y) $G_{u,v}$ — это значение которое содержит в себе пиксель с координатами (u, v)

После преобразования можно заметить, что значение $G_{0,0}$ превосходит все остальные, его так же называют *коэффициентом DC*. Он определяет основной тон для блока в целом. Его так же можно назвать *постоянной компонентой*. Оставшиеся 63 коэффициента (блока 8x8) называют *AC коэффициентами*, где AC могут быть установлены для запасных комп онент. Преимущество дискретного косинусного преобразования — возможность вычислить основной оттенок блока (сигнала).

Квантование

Человеческий глаз хорошо приспособлен замечать маленькие различия в яркости на относительно больших расстояниях, но плохо отличает точную силу яркости на высоких частотах. Это позволяет значительно уменьшить количество информации о высокого частотных компонентах. Данную операцию можно осуществить простым делением значения каждой компоненты в частотном диапозоне на константу и последующим округлением этого значения ближайшим целым числом. Это действие, единственное во всем процессе сжатия при котором происходит потеря данных, в отличии от дискретного косинусного пребразования, которое выполняет вычисления с выской точностью. Как результат, многие компоненты оказваются равны нулю или их значение очень близко к нулю, как следствие они занимают меньше бит в памяти.

Подобные процессы, когда процедура построения чего-либо ведется с помощью набора дискретных (в нашем случае целых) величин, называется *квантованием*.

Элементы *матрицы квантования* управляют коэфициентом сжатия. Чем больше значение компонент, тем больше будет потеря изображения в качетсве. Типичная матрица квантования (качество ухудшается примерено на 50%, утверждена как часть стандарта JPEG), выглядит следующим образом:

$$Q = \begin{bmatrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{bmatrix}.$$

Квантование коэффициентов матрицы G, полученной на предыдущем шаге, происходит следующим образом: e

$$B_{j,k} = \text{round}\left(\frac{G_{j,k}}{Q_{j,k}}\right) \text{ для } j = 0, 1, 2, \ldots, 7; k = 0, 1, 2, \ldots, 7$$

Полученная матрица B, отправляется на следующий шаг, где будет произвоидиться ее энтропийное кодирование.

Энтропийное кодирование

Энтропийное кодирование — кодирование последовательности значений с возможностью однозначного восстановления с целью уменьшения объёма данных (длины последовательности) с помощью усреднения вероятностей появления элементов в закодированной последовательности.

В JPEG формате реализация включает в себя переопределение порядка компонент изображения в виде *"зиг-зага"* (рис. 1) и дальнейшее сжатие методом *кодирования повторов*, который группирует похожие повторяющие значения вместе. Затем к получившейся последовательности применяется *Код Хаффмана*.

Для описания процесса сжатия данных представленных в "зиг-заг" порядке - кодирования повторов, введем некоторые определения:

- x — ненулевой AC коэфициент, получившийся после квантования;
- $RUNLENGTH$ — число нулей перед ненулевым AC коэффициентом;
- $SIZE$ — число бит, требуемых чтобы представить x;

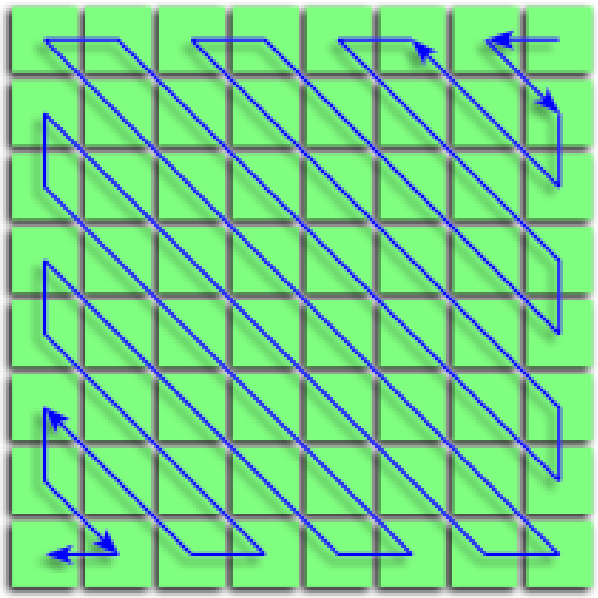

Рис. 1: Порядок "зиг-заг" в котором кодируются компонеты блока.

- $AMPLITUDE$ — побитовое представление x.

Кодирование повторов работает исследуя каждый ненулевой AC коэффицинт x и определяя, как много нулей следует перед ним. С этой иноформацией создаются два *символа*:

$$\begin{bmatrix} Symbol1 & Symbol2 \\ RUNLENGTH, SIZE & AMPLITUDE \end{bmatrix}.$$

Оба числа RUNLENGTH и SIZE хранятся в одном и том же байте, под каждый из них отводится по 4 бита. Так значения любого из них не превосходит 64.

Так же в стандарт JPEG определены специальные символы, которые означают окочание блока (EOB), и еще один на случай? когда встречаются более более 15 нулей подряд, до достижения ненулевого коэффицинта, тогда символ кодируется специально как: (15, 0) (0).

Весь процесс продолжается до тех пор? пока не будет достигнут символ окончания блока. Определенный как символ EOB — (0,0).

1.1.2. Декодирование

Декодирование для отображения изображения состоит из тех же шагов, выполняемых в обратном порядке.

Возьмем раскодированую матрицу коэффициентов, полученную после примения к ней дешифрования кода Хаффмана и декодирование повторов. И расположим символы этой последовательности в правильном порядке. В результате получится матрица B, являющуюся результатом шага квантования. Затем умножим матрицу B на матрицу квантования Q по Адамару. В результате получим матрицу F

$$F = B \circ Q,$$

которая очень похожа на оригинальную матрицу G, явяляющуюся результатом дискретного косинусного преобразования.

Следующим шагом является применение обратного косинусного преобразования, которое можно описать следюущим образом:

$$f_{x,y} = \frac{1}{4} \sum_{u=0}^{7} \sum_{v=0}^{7} \alpha(u)\alpha(v) F_{u,v} \cos\left[\frac{(2x+1)u\pi}{16}\right] \cos\left[\frac{(2y+1)v\pi}{16}\right],$$

где

- x столбец пикселей $0 \leq x < 8$;
- y строка пикселей $0 \leq y < 8$;
- $F_{u,v}$ это компонента матрицы коэффициентов полученая на предыдущем шаге с координатам (u, v);
- $f_{x,y}$ — значение полученой компоненты с координатами (x, y).

Затем значение каждой компоненты округляются до ближайшего целого числа, так как изначально значение компоненты целочисленное, и к каждому значению прибавляется 128. На этом процедура декодирования заканчивается. И мы получаем изображение практически идентичное оригиналу.

1.2. Сжатие JPEG изображений без потерь

Как мы могли заметить, в основе оригинального алгоритма JPEG лежит сжатие по коду Хаффмена. Как известно, за последнее время появилось множество других алгоритмов энтропийного кодирования, которые гораздо эффективнее компрессируют информацию в сравнении с данным алгоритмом. Это повлекло за собой образование целого класса алгоритмов, которые позволяют нам дальнейшее сжатие JPEG изображений без потери в качестве.

Одним из представителей этого класса алгоритмов является программа packJPG, в основе которой лежит алгоритм с одноименным названием. Преимуществом данного алгоритма является высокая степень сжатия,а недостатком — низкая скорость кодирования, в сравнение с алгоритмами подобного рода.

До недавнего времени код алгоритма был закрыт, но в 2014 году был опубликован и получил лицензию GPL3, что позволило в данной работе проанализировать его исходный код, понять основные идеи алгоритма и его реализации.

1.3. Технология OpenMP

В ходе данной работы проведена попытка адаптировать исходный алгоритм packJPG для многопроцессорных архитектур и создать его более эффективную реализацию. Код исходного проекта написан на языке C++ и поэтому для реализации была выбрана технология OpenMP, которая зарекомендовала себя как простой и удобный инструмент для многопоточного программирования. Важным достоинством технологии OpenMP является возможность реализации так называемого инкрементального программирования, когда программист постепенно находит участки в программе, содержащие ресурс параллелизма, с помощью предоставляемых механизмов делает их параллельными, а затем переходит к анализу следующих участков. Таким образом, в программе нераспараллеленная часть постепенно становится всё меньше. Такой подход значительно облегчает процесс адаптации последовательных программ к параллельным компьютерам, а также отладку и оптимизацию.

2. Алгоритм packJPG

PackJPG — это пограмма специально разработанная для дальнейшего сжатия JPEG изображений без потерь. Обычно ей удается уменьшить размер файла JPEG изображения примерно на 20%.

В данном подходе JPEG файл первоначально декодируется в коэффициенты, получившиеся после шага квантования. Далее эти коэффициенты перегруппируются в 64 изображения для каждой цветовой компоненты, содержащейся в исходном изображении. Каждое сгруппированное изображение содержит все коэффициенты 'подзоны' в соответствии с двумерной базисной функции дискретного косинусного преобразования(DCT коэффициенты).

Обычно существуют статистические зависимости между соответсвующими DCT коэффициентами соседних блоков — в стандартном алгоритме JPEG эта избыточность, в некотором роде, подавляется только лишь дифференциальным кодированием коэффициентов 'подзоны'.

Особенностью данного подхода являются следующие шаги, которые применяются к раскодированным коэффициентам: Paeth предиктор, использование списков концов блоков, оптимизированное сканирование и арифметическое кодирование, как один из вариантов энтропийного кодирования. Рассмотрим каждый их них более детально.

2.1. Paeth предиктор

Раскодированные и сгруппированные коэффициенты представляют собой уменьшенную копию исходного изображения. Следовательно,к ним можно применить те же техники декорреляции, что используются для изображений в градации серого. Одной из самых простых и эффективных подоюных техник является Paeth предиктор, который описан в спецификации формата PNG. Его предназначение в следующем: он пытается выбрать лучшее возможное направление для каждого пикселя. Вычисляется простая линейная функция из трех соседних пикселей (левый, верхний, левый-верхний), выбирается тот пиксель, который ближе всего к результату.

2.2. Использование списков концов блоков

В макроблоке из 64 коэффициентов, отсортированных в зиг-заг порядке, *конец блока* определяется, как позиция после последнего ненулевого коэффициента. Все концы блоков каждого макроблока из одной цветовой компоненты группируются вместе в форме списка концов блоков для этой цветовой компоненты.

В стандартном алгоритме сжатия JPEG большинство коэффициентов после шага квантования принимают нулевые значения и на самом деле большая часть из них встречается в конце зиг-заг упорядоченных коэффициентов. Следовательно, концы

блоков могут быть использованы для эффективной группировки и кодирования последних нулей. Такой же подход применятся в стандартном алгоритме JPEG, где не используются списки концов блоков. Там концы блоков кодируются с помощью специального символа.

В подходе packJPG вышеуказанные списки используются для дополнительной цели. Можно предположить, что чем позже появляется конец блока, то тем больше высокочастотных компонент содержится в макроблоке.

В данном подходе блоки 8x8 группируются в соответсвии со значениями концов блоков. Каждая группа блоков кодируется независимо от других групп. Оптимальное количество уровней варьируется от изображения к изображению. Это зависит от размера изображения. Границы уровней вычисляются через линейные квантили списка концов блоков.

2.3. Оптимизированное сканирование

Коэффициенты кодируются для "подзоны", коэффициент за коэффициентом, используя статистическую модель первого порядка. Но в отличии от стандартного алгоритма в packJPG обход компонент блока делается не в прямом порядке, а в специальном.

Так например, обычный горизонтальный обход применяется только для компонент лежащих ниже диагонали блока. Компоненты лежащие выше диагонали кодируются с помощью вертикального обхода. А все элементы диагонали блока кроме DC коэффициента кодируются в порядке основанном на на нескольких кривых Гильберта.

2.4. Арифметическое кодирование

В данном подходе используются статистические модели разных порядков для кодирования разных типов данных с помощью алгоритма *арифметического кодирования.*

Так например, JPEG заголовок, который необходим для сжатия без потерь, будет использовать статистическую модель первого порядка.

Списки конца блоков будут компрессироваться с помощью двуразмерной статистичкской модели. Для каждого значения конца блока в качесве контекста статистической модели используются значения верхнего и левого блоков.

Коэффициенты, получившиеся после дискретного косинусного преобразования, кодируются в соответсвии со знаком значения, его абсолютной величиной и категорией. Эту схему можно объснить следующим образом.

В большинстве случаев знак коэффициентов не может быть сжат ниже чем до 1 бита на коэффициент. Используя статистическую модель первого порядка и оптимизированный порядок обхода компонент, описанный выше, их сжатый размер не будет

превышать одного бита на знак.

Оставшиеся абсолютные значения коэффициентов дискретного косинусного преобразования кодируются с помощью схемы очень похожей на ту, что приведена в стандарте JPEG под названием *Variable Length Integer*. Сначала значения разбиваются по категориям, после обхода сгруппированных коэффициентов с помощью оптимизированного порядка, мы будем применять для кодирования последовательности статистическую модель первого порядка, используя предшествующие значения как контекст для модели.

Полученные коэффициенты отправляются на еще один этап сжатия, где будут сжиматься опять таки с помощью статистической модели первого порядка, но в качетсве контекста уже будет использоваться категория. Значения категорий были вычислены империческим путем.

3. Анализ возможности реализации с помощью технологии OpenMP

Анализ возможности реализации включает в себя следующие этапы: анализ существующего кода приложения *packJPG*, рефакторинг существующего кода, поиск узких мест алгоритма, модификация с помощью технологии OpenMP, оценка эффективности осуществленной модификации. Рассмотрим каждый из них более подробно.

3.1. Анализ существующего кода приложения

Исходный код приложения состоит более чем из 10000 строк, и этот достаточно внушительный объем кода содержится в 5 файлах. Поэтому значительная часть времени была затрачена на изучнение структуры приложения и анализ деталей реализации алгоритма *packJPG*.

Основной код приложения расположен в файле *packjpg.cpp*. PackJPG является полноценным программным обеспечением для сжатия JPEG изображений, и большая часть исходного кода предназначена для предоставления удобного контроля за функционалом программы. Так для *Windows* и *MacOS* написана реализация графического пользовательского интерфейса, а также для всех платформ реализован *интерфейс командной строки*.

Эта часть программа, имеет малую вычислетельную сложность и не относится к алгоритмам компрессии. Поэтому будут рассмотрены компоненты кода, которые ответсвенны за сам процесс сжатия.

3.1.1. JPEG кодирование и декодирование

Весь код, ответственный за кодировние и декодирование сжатия формата JPEG, содержится в двух основных методах. Так процедурой по извлечению всей структуры JPEG изображения, представленной в главе 1, занимается метод *decode_jpeg* , а за обратную процедуру отвечает метод *recode_jpeg* .

3.1.2. Предиктивное кодирование

Данная релизация поддерживает несколько вариантов линейного предиктивного кодирования, описанного в п. 2.1. Присутствует возможность опционально выбирать тип предиктивного кодирования для лучшего сжатия. Первая реализация — это *loco-i* предиктор и ответсвенный за него метод: *int plocoi(int a, int b, int c)*, второй — *dc_coll_predictor*, предиктор, используемый для массива данных и реализующий его метод: *int dc_coll_predictor(int cmp, int dpos)*. Третья реализация — предиктор, ис-

пользуемый для DC коэффициентов, базирующийся на дискретном косинусном преобразовании: *int dc_1ddct_predictor(int cmp, int dpos)*

3.1.3. Арфимитическое кодирование

Основным компонентом определяющим интерфейс и являющимся кодировщиком и декодировщиком последовательности *символов* является класс *aricoder* со следующими публичными методами, представленными на листинге:

```
class aricoder {
public:
    aricoder(iostream *stream, int iomode);

    ~aricoder(void);

    void encode(symbol *s);

    unsigned int decode_count(symbol *s);

    void decode(symbol *s);
    ...
```

Под *символом* понимают следующую структуру, которая хранит информацию о вероятностном диапозоне кодируемого числа и его частоте, код структуры приведен в следующем листинге:

```
struct symbol {
    unsigned int low_count;
    unsigned int high_count;
    unsigned int scale;
};
```

Основным классом, который используется для энтропийного кодирования, в случае арифметического кодирования, является статистическая модель. В данном коде существует две модификации: *model_s* — модель первого порядка и *model_b* — двумерная модель. Они имеют схожий интерфейс. Например класс *model_s* имеет следующую структуру:

```
class model_s {
public:

    model_s(int max_s, int max_c, int max_o, int c_lim);

    ~model_s(void);

    void update_model(int symbol);

    void shift_context(int c);
```

```
    void flush_model(int scale_factor);

    void exclude_symbols(char rule, int c);

    int convert_int_to_symbol(int c, symbol *s);

    void get_symbol_scale(symbol *s);

    int convert_symbol_to_int(int count, symbol *s);

    bool error;


private:

    // unsigned short* totals;
    unsigned int *totals;
    char *scoreboard;
    int sb0_count;
    table_s **contexts;
    table_s **storage;

    int max_symbol;
    int max_context;
    int current_order;
    int max_order;
    int max_count;
```

Для хранения контекста моделей используются специальные таблицы — это структуры данных, содержащие следующую информацию: счетчики для каждого символа, содержащегося в таблице; ссылки на таблицы порядком выше и ниже; накопительные счетчики. Листинг структуры *table*, используемой в *model_ b*, будет привиден ниже. Похожая реализация имеется и для класса *model_ s* и описана в структуре под названием *table_ s*:

```
struct table {
    // counts for each symbol contained in the table
    unsigned short *counts;
    // links to higher order contexts
    struct table **links;
    // link to lower order context
    struct table *lesser;
    // accumulated counts
    unsigned int scale;
};
```

3.2. Рефакторинг существующего кода

Как говорилось в предудущей главе, код проекта плохо поддавался анализу и требовал большого времени для поиска, той или иной компоненты. Еще одной особенностью кода данного приложения являлется его процедурный стиль. К основным недостаткам также можно отнести длинные методы, многократное дублирование кода, расходящиеся модификации.

Очевидно, что данный код требовал рефакторнинга, который был произведен. Основные проблемы и частные случаи их решения будут рассмотрены в следующих пунктах.

3.2.1. Реструктуризация исходного кода

Первым этапом в подготовке кода к дальнейшей разработке стала декомпозиция больших файлов и выделение оттуда компонент с определенной зоной ответственности. Компоненты были разбиты на группы по области применения и распределенны по соответствующим пакетам. Например, такие классы как *model_ s*, *model_ b* и др. оказались в пакете *aricoder*, ответсвенном за арифмитичское кодирование.

Так же были выделены и сформированы пакеты *gui* и *utils*, отвечающие за графический пользовательский интерфейс и различные утилитные классы.

Первоначальная сборка проекта осуществлялась с помощью обычного *Makefile*, что осложняло ее рутинными процедурами и не позволяло гибко управлять ею. Одной из задач была интеграция библиотки OpenMP в данное приложение, но стало бы мешать другим разработчикам, у которых непредустановлена эта библиотека в системе,вынуждая заниматься ее инсталяцией. Поэтому для сборки был выбран такой нструмент как *CMake*, для которого уже реализован модуль *FindOpenMP*. Он отвечает за поиск библиотеки OpenMP, а в случае, если она не обнаружена — директивы данной библиотеки игнорируются.

3.2.2. Дублирование кода

Дублирование кода — это случай, когда два фрагмента кода выглядят почти одинаково. В данном исходном коде такого рода дублирования достаточно много. Это понижает читабельность кода и делает его структуру более сложной, ветвистой и непонятной.

Например, код процедуры сортировки массива встречается в нескольких местах и представляет собой обычную *сортировку вставками*. Для избавления от дублирования в данном случае было использовано извлечение метода или просто перенос процедуры сортировки в один из утилитных классов. Так в класс *common_ utils* был извлечен метод *insertion_ sort*, который можно увидеть в листинге приведенном ниже:

```
class common_utils
```

```
{
    ...
    template <class T>
    static void insertion_sort(T* values, int size )
    {
        bool done;
        T swap;
        int i;

        // sort data first
        done = false;
        while ( !done ) {
            ...
        }

    }
    ...
}
```

3.2.3. Длинные методы

Еще одной особенностью данного исходного кода является высокая цикломатическая сложность некоторых методов и длина их кода. Так например, методы *decode_jpeg* или *encode_jpeg* имеют длину более 300 строк. Это не является целесообразным, и делает такой код очень трудночитаемым и плохо переиспользуемым. Такие участки кода необходимо декомпозировать. Для разрешения этой проблемы была проведена декомпозиция кода, извлечены методы с определенными обязанностями, переменные были заменены вызовами методов.

3.3. Модификация с помощью технологии OpenMP

3.3.1. Поиск узких мест программы

В предыдущих пунктах этой главы было замечено, что программа имеет большое количество процедур и кода, следовательно первым делом возникла идея парллельной реализации тех методов, которые исполняются чаще всего. Таким образом, распараллеливание этих методов могло дать максимальный выигрыш в производительности.

Для оценки времени исполнения кода каждого метода использовался профайлер *valgrind*, который имеет бесплатную лицензию, гибкую настройку и обладает всем требуемым функционалом. На основе результатов профайлера было установлено, что самой долгой по времени выполнения является шаг арифмитеческого кодирования последовательности элементов. Дольше всего алгоритм находится в методах *shift_context* и *update_model* классов статистических моделей *model_s* и *model_b*.

3.3.2. Параллельные модификации

Была произведена параллельная реализация нескольких десятков участков кода.Рассмотрим несколько примеров таких модификаций более подробно. При первоначальной оценке данного алгоритма предполагалось, что удастся добиться высокого уровня распараллеливания за счет блочных структур, которые активно задействованы в формате JPEG. Следовательно в исходном коде часто используется такая конструкция как цикл, что является благоприятным условием для многопоточного программирования.

Одним из самых популярных видов распараллеливания стало распараллеливание существующих циклов *for*. Так например неэффективная *сортирвка вставками*:

```
template <class T>
    static void insertion_sort(T* values, int size )
    {
        bool done;
        T swap;
        int i;
        done = false;
        while ( !done ) {
            done = true;
            for ( i = 1; i < size; i++ )
                if ( values[ i ] < values[ i - 1 ] ) {
                    swap = values[ i ];
                    values[ i ] = values[ i - 1 ];
                    values[ i - 1 ] = swap;
                    done = false;
                }
        }

    }
```

была заменена параллельной реализацией быстрой сортировки, что дало прирост примерно в 2%. Фрагмент данной модификация отражен на следующем листинге:

```
template<class T>
    static void sort(T *a, int n) {
        long i = 0, j = n;
        float pivot = a[n / 2];
        do {
            while (a[i] < pivot) i++;
            while (a[j] > pivot) j--;
            if (i <= j) {
                std::swap(a[i], a[j]);
                i++;
                j--;
            }
```

```
        } while (i <= j);
        if (n < 100) {
            if (j > 0) sort(a, j);
            if (n > i) sort(a + i, n - i);
            return;
        }
        #pragma omp task shared(a)
        if (j > 0) sort(a, j);
        #pragma omp task shared(a)
        if (n > i) sort(a + i, n - i);
        #pragma omp taskwait
    }

};
```

Так как одним из этапов алгоритма является выполнение дискретного косинусного преобразования и обратного дискретного косинусного преобразования. Многие процедуры связаны с транформацией блоков или же с их обходом, поэтому эти участки содержат элементарные циклы которые хорошо поддаются распараллеливанию. Примером может служить метод используемый на этапе обратного косинусного преобразования *int idct_2d_fst_8x8(signed short* F, int ix, int iy)*:

```
inline int idct_2d_fst_8x8( signed short* F, int ix, int iy )
{
        int idct;
        int ixy;
        int i;

        ixy = ( ( iy * 8 ) + ix ) * 64;

        idct = 0;
        for ( i = 0; i < 64; i++ )
                idct += F[ i ] * icos_idct_8x8[ ixy++ ];

        return idct;
}
```

И после паралельной модификации используя параллелизм циклов и опцию редукции библиотеки openMP, была получена следующая модификация.

```
int idct_2d_fst_8x8( signed short* F, int ix, int iy )
{
        int idct;
        int ixy;
        int i;
        ixy = ( ( iy * 8 ) + ix ) * 64;
        idct = 0;
    #pragma omp parallel for reduction(+:idct)
```

```
        for ( i = 0; i < 64; i++ )
                idct += F[ i ] * icos_idct_8x8[ixy + i];
        return idct;
}
```

3.3.3. Проблемы модфикации

Основной проблемой, которая были обнаружена во время реализации — это то, что самым трудоемким участком кода являются методы связанные с обновлением и обработкой статистической модели. Ярким примером является метод *shift_context*, приведенный в данном листинге.

```
void model_s::shift_context( int c )
{
    table_s* context;
    int i;

    // shifting is not possible if max_order is below 1
    // or context index is negative
    if ( ( max_order < 1 ) || ( c < 0 ) ) return;

    // shift each orders' context
    for ( i = max_order; i > 0; i-- ) {
        // this is the new current order context
        context = contexts[ i - 1 ]->links[ c ];

        // check if context exists, build if needed
        if ( context == NULL ) {
            // reserve memory for next table_s
            context = ( table_s* ) calloc( 1, sizeof( table_s ) );
            if ( context == NULL ) ERROR_EXIT;
            // set counts NULL
            context->counts = NULL;
            // setup internal counts
            context->max_count  = 0;
            context->max_symbol = 0;
            // link lesser context later if not existing, this is done
               below
            context->lesser = contexts[ i - 2 ]->links[ c ];
            // finished here if this is a max order context
            if ( i == max_order )
                context->links = NULL;
            else {
                // build links to higher order tables otherwise
                context->links = ( table_s** ) calloc( max_context,
                   sizeof( table_s* ) );
                if ( context->links == NULL ) ERROR_EXIT;
```

```
                // add lesser link for higher context (see above)
                contexts[ i + 1 ]->lesser = context;
            }
            // put context to its right place
            contexts[ i - 1 ]->links[ c ] = context;
        }

        // switch context
        contexts[ i ] = context;
    }
}
```

Как можно увидеть, этот метод требует строго последовательного выполнения, то есть код выполняется связно, и следующий результат зависит от предыдущего. Такие участки кода практически не поддаются распараллеливанию, и единственным вариантом стало введение параллельных секций внутри тела цикла, которые работают с независимыми участками кода:

```
#pragma omp parallel {
    context->max_count  = 0;
    context->max_symbol = 0;
    // link lesser context later if not existing, this is done below
    context->lesser = contexts[ i - 2 ]->links[ c ];
    // finished here if this is a max order context
    if ( i == max_order )
        context->links = NULL;
    else {
        // build links to higher order tables otherwise
        context->links = ( table_s** ) calloc( max_context, sizeof(
            table_s* ) );
        if ( context->links == NULL ) ERROR_EXIT;
        // add lesser link for higher context (see above)
        contexts[ i + 1 ]->lesser = context;
    }
}
```

Однако, после проведения тестов выяснилось, что такая модификация не приносит положительного эффекта. И данный метод пришлось оставить без модификаций.

4. Результаты

Для оценки внесенных модификаций, были проведены тесты производительности. Тесты проводились на компьютере с процессором Intel Celeron 1007U, имеющим два физических и два вирутальных ядра с частотами 1.5 GHz. Сравнительная таблица результов работы до и после модификации представлена ниже:

Размер изображения (пикс./мегабайт)	Исходная реализация (сек.)	Параллельная модификация (сек.)
400x400/0.0136	0.027	0.021
1920x1080/1.075	1.387	1.317
4000x4000/3.500	5.22	5.01

В среднем был получен прирост в производительности приблизительно на 3-4%, что является приемлимым, но не оптимальным результатом. Данный результат связан прежде всего с высокой связностью алгоритма packJPG, где многие итеративные элементы кода не поддаются полному распараллеливанию.

Заключение

В настояещей дипломной работе был проведен анализ возможности и эффективности параллельной реализации на примере алгоритма packJPG, являющегося компонентом одноименной программы. В процессе выполнения работы был изучен стандарт изображений JPEG и его представление, кодирование и декодирование. Также осуществлен анализ алгоритмов сжатия JPEG без потерь и выявлены основые преимущества и недостатки алгоритма packJPG.

В ходе основного этапа работы был осуществлен анализ кода и выявлены его основноые компоненты. Данная процедура потребовала массивного рефакторинга существующего кода. На основе этого было выполнено более детальное описание алгоритма.

Параллельная реализация осуществлялась с помощью техники инкременатального программирования. Процедура реализации выполнялась итеративно.Итерация представляла из себя параллельную реализацию фрагмента кода и оценку эффективности этой модификации. В некоторых частях кода ее произвести оказалось невозмоожным, в связи со спецификой программы.

Проведенная работа может быть продолжена в виде реализации данного алгоритма для графических вычислительных устройств, так как код наполенен большим количеством циклов, работающих с одноразмерными и двуразмерными векторами. Видеокарта имеет болшое количеством ядер для вычисления, что должно увеличить эффективность проведеной реализации, если ее удастся адапировать для видеокарт.

Список литературы

[1] Mitchell William B. Pennebaker; Joan L. JPEG still image data compression standard. — 1993. — P. 290–293.

[2] Stirner Matthias, Seelmann Gerhard. IMPROVED REDUNDANCY REDUCTION FOR JPEG FILES. — 2007.

[3] А.С. Антонов. Параллельное программирование с использованием OpenMP. — Изд-во МГУ, 2009.

[4] The JPEG Specification // `http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=59634`. — 2012.

[5] The PNG Specification // `https://www.w3.org/TR/PNG-Structure.html`. — 1996.

[6] The official packJPG site // `http://packjpg.encode.ru`.

[7] The official OpenMP site // `http://www.openmp.org`.

[8] The official CMake site // `https://cmake.org`.

[9] The official valgrind site // `http://valgrind.org`.